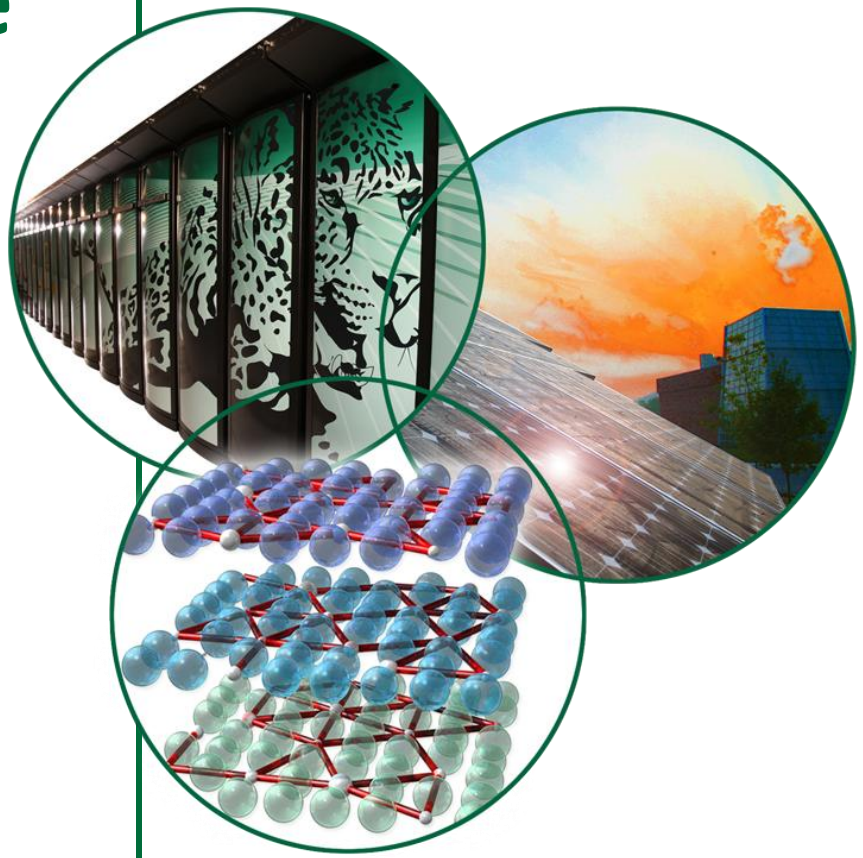
Advanced VampirTrace



Tips & Tricks

Thomas Ilsche, ORNL

May 17, 2011

U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Overview

- VampirTrace on lens etc.
- Write a runtime filter file
- Limit compiler instrumentation
- Manual source code instrumentation
- Merge / Shrink large trace files
- Additional topics

Documentation

- Remember that the documentation is very valuable!
- /sw/sources/vampirtrace/5.11ornl/doc/UserManual.pdf
- Instrumentation
  - Note: Dyninst and Library tracing is not supported on Jaguar
- Runtime Options
- Counters
- Filtering & Grouping
- Command Reference

VampirTrace on lens etc.

- On normal Linux systems `gcc` (or the PE-specific compiler) is the default compiler for `vtcc`
  →For MPI, tell vtcc to use mpicc

```
$vtcc -vt:cc mpicc
$vtcxx -vt:cxx mpicxx
$vtf90 -vt:f90 mpif90
```

- If your build system uses `gcc -lmpi` manually, you do not need to specify `mpicc`

GPU Tracing

- Works on yona or lens
  - CUDA version on lens is old
- Library calls to cudart are wrapped, trace contains:
  - API Calls
  - CUDA streams as ‘Threads’
  - Kernel execution as function execution on the streams
  - Memory copy as ‘MPI p2p’
  - CUPTI performance counters
- Environment variables
  - VT_CUDARTTRACE=yes (set by `module load`)
  - VT_CUPTI_METRICS=local_store:local_load

Write a runtime filter File

- Motivation
  - Reduce trace size
  - Reduce flush overhead
- Filter file format

```
#     call limit definitions and region assignments
#     syntax: <regions> -- <limit>
#       regions     semicolon-separated list of regions
#                   (can be wildcards)
#       limit       assigned call limit
#                    0 = region(s) denied
#                   -1 = unlimited
add;sub;mul;div -- 0
very_important_function -- -1
unintersting_module* -- 0
* -- 3000000
```

Write a runtime filter File (cont.)

- Apply using `export VT_FILTER_SPEC=filter.txt`
- `vtfilter` generates filter file from traces
  - `$ vtfilter --gen -r 50 -stats \`
    `-o filter.txt intput.otf`
  - Parallel version for large traces
    `$ aprun -n 1044 vtfilter-mpi -o ./filter.out`
    `--reduce=20 -s ./trace.otf`
    - Doesn't need as many processes as the original application, try 1/32

Limit compiler instrumentation (gcc)

- Motivation
  - Compiler instrumentation is easy to use
  - But runtime filters do not eliminate the complete overhead
  - And manual instrumentation is time-consuming
- Two parameters for gnu compilers
  - `-finstrument-functions-exclude-function-list`
  - `-finstrument-functions-exclude-file-list`
  - Arguments separated by comma
  - Match is done by substrings, but no real wildcards
  - User-visible name is used (rather than mangled one)
- Strange behavior for inlined functions
- Global constructors appear to be ‘resistant’

Limit compiler instrumentation (C)

- Alternative solution using code modification in C
  - `__attribute__ ((__no_instrument_function__))` before a function declaration
  - Works for gcc, icc, pathcc
  - Does not affect pgcc

Source instrumentation using TAU/PDT

- Motivation
  - Allows selective automatic source code instrumentation
  - No problems with internal compiler functions (e.g. global constructors) polluting the trace
- Add `-vt:inst tauinst -vt:tau "-f <filename>"`

```
BEGIN_FILE_EXCLUDE_LIST
*stl_tree.h
*.I
END_FILE_EXCLUDE_LIST
BEGIN_EXCLUDE_LIST
check_foo_#
END_EXCLUDE_LIST
```

  - Also allows include lists
- Some issues with compiler specific codes (e.g. Macros)

Manual Source Code instrumentation

- Motivation
  - Best control over instrumentation
  - Also record non-function regions or non-PAPI counters
- API available for C and Fortran.
- Instrumentation is done by using macros, so it can be disabled without any impact on the application

Manual Source Code instrumentation (C)

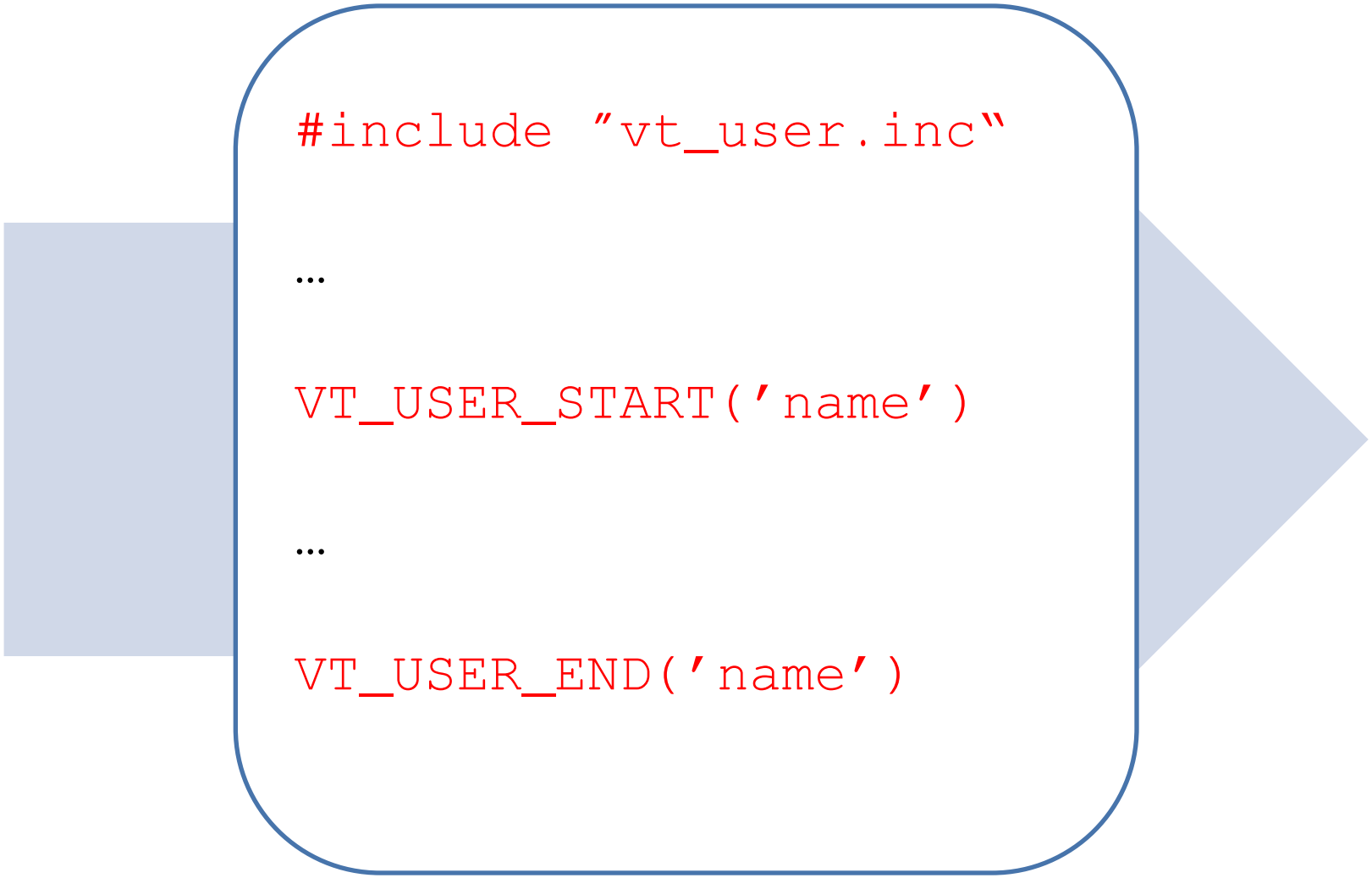
```
#include "foo.h"
void bar() {
  compute1();
  if (cond()) {
    return;
  }
  compute2();
}
```

```
#include "foo.h"
#include "vt_user.h"
void bar() {
  VT_USER_START("bar");
  compute1();
  if (cond()) {
    VT_USER_END("bar");
    return;
  }
  compute2();
  VT_USER_END("bar");
}
```

Manual Source Code instrumentation (C++)

```
void run() {
  for (i=0;i<nSteps;i++) {
    VT_TRACER("timestep");
    compute1();
    compute2();
    compute3();
  }
}
```

Manual Source Code instrumentation (Fortran)



Manual Source Code instrumentation

- Needs to be explicitly enabled during compilation
- `$ vtcc -vt:inst manual -DVTRACE hello.c -o hello`
- Otherwise there will be no overhead
- Source code instrumentation can be combined with all other instrumentation types

Merge large trace files

- otfmerge-mpi reduces the number of files in a trace while keeping all processes and events

```
$ aprun -n 10000 otfmerge-mpi -n 10000 -z 9 -o trace-merged trace.otf
```

- Huge traces can become more easy to handle
- Use only when you are stuck or trace handling is inconvenient

Shrink large trace files

- otfshrink selects / hides processes
- Select a list of processes:

```
$ otfshrink -i input.otf -o shrink.otf -l 1 10 100 1000
```

- Select a range of processes (first 100):

```
$ otfshrink -i input.otf -o shrink.otf -l 1-100
```

- Select every 4th process of 4096:

```
$ otfshrink -i input.otf -o shrink.otf -l `seq 1 4 4096`
```

- Created trace is just a collection of symlinks
- Huge traces can become more easy to handle
- You might lose the view on performance anomalies (especially MPI) that correlate with your selection pattern

Shorten long function names

- `$ vtbeautify foo.otf`

```
#!/bin/bash
DEF=$1.0.def
echo ”Beautifying $DEF, backup in $DEF.ugly.“
cp $DEF.z $DEF.z.ugly
otfdecompress $DEF.z
cp $DEF $DEF.ugly
cat $DEF | sed ’s/[(].*[)]//’ | sed ’s/<.*>//g’ |
| sed ’s/__/_/’ | sed ’s/_module_MOD//’ > $DEF
```

- Removes C++ function and template-parameters
- Removes _module_MOD from Fortran names
- Removes double underscore

Additional topics (1)

- Additional source code instrumentation
  - User defined Counters
    - Keep track of numerical values during program execution, e.g. Buffer sizes
  - User defined Markers
    - Add arbitrary textual information to the trace
    - More convenient to browse in a visual timeline vs. textual logging
- Function grouping gives a better overview

Additional topics (2)

- Separate trace unification
  - Useful for time measurements on aprun
  - ```
    $ export VT_UNIFY=no
    $ aprun -n 4096 ./MyApplication
    $ aprun -n 1024 vtunify-mpi MyApplication
    ```
- Add line number information to the trace
  - ```
    $ export VT_GNU_NM=/tmp/work/$USER/.vt/bin/nm \
      --demangle --line-numbers
    ```
  - Can take long time at application startup
  - ```
    $ nm --demangle --line-numbers foo > foo.nm
    $ export VT_GNU_NMFILE=foo.nm
    ```
    ```

Thank you

- Contact:
  - Thomas Ilsche, ORNL
    5700 B206
    tt1@ornl.gov
    865-241-6293